John von Neumann Institute for Computing


# Exploiting Parallelism on Irregular Applications Using the GPU

M. Ujaldón, J.H. Saltz

published in

*Parallel Computing:*
*Current & Future Issues of High-End Computing*,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 639-646, 2006.



http://www.fz-juelich.de/nic-series/volume33

# Exploiting parallelism on irregular applications using the GPU *

Manuel Ujaldon[a], Joel Saltz[b]

[a]Computer Architecture Department. University of Malaga. Complejo Tecnologico. Campus Teatinos. Malaga, 29071. SPAIN

[b]Biomedical Informatics Department. Ohio State University. 3197 Graves Hall. 333 W. 10th Ave. Columbus, Ohio 43210. U.S.A.

The computational speed on microprocessors is increasing faster than the communication speed, especially on parallel processors such as GPUs. Thus, the computations that benefit the most from GPU processing have high arithmetic intensity. This paper compares the effectiveness of GPUs when handling scientific general-purpose irregular problems, outperforming counterpart CPUs by a wide margin and identifying the AGP bus as the major bottleneck in graphics architecture. We study the impact that the emerging PCI-Express bus has for accelerating such applications when replacing AGP. A number of software optimizations are also conducted by using recent APIs, OpenGL extensions and drivers, leading to loading times 40% lower on PCI-Express and four times faster when overlapping communication with computation. Execution times are shown on a benchmark composed of an Euler solver and a sparse matrix-vector product running on Nvidia GeForce FX and GeForce 6800 GT graphics cards.

## 1. Introduction

By taking advantage of the streaming processing model, modern graphics processors (GPUs) are outperforming their CPU counterparts in some general-purpose applications, and the difference is expected to grow in the future [7].

Modern CPUs have been increasing their performance according to Moore's Law over the last three decades. Such improvements have been mostly based on clock frequency and transistors manufacturing process, which find severe boundaries for progressing in the future. GPUs, on the contrary, double performance every six months relying on memory latency rather than on raw speed. Their improvements are focused on architectural layers, by setting a streaming execution model which reverses the bottleneck inherent to memory access: Data are the axis flowing through the graphics pipeline, and instructions are those who come to meet them. Since there is no memory hierarchy nor data dependencies in the streaming model, the pipeline maximizes throughput without being stalled. That way, whenever the GPU is consistently fed by input data, performance boosts, leading to an extraordinary scalable architecture.

We have taken advantage of these extraordinary capabilities by developing methods for mapping irregular general-purpose algorithms onto the GPU [12,13], where we outperform CPU performance by a 2x-4x factor in execution time. Nonetheless, a major bottleneck located in the AGP bus affected performance when accounting the time for loading the input data onto the GPU.

This paper contributes to overcome such bottleneck by exploring the features of the PCI-Express bus recently introduced for the graphics cards in commodity PCs, and performing a number of optimizations using OpenGL extensions. Other issues regarding data communication and accessing

*This work was partially supported by the Ministry of Education of Spain, through the Secretary of State for Education and Universities, grant PR2004-0508.

| Graphics processing | Conventional programming | Graphics processing | Conventional programming |
|---|---|---|---|
| Texture memory | Arrays in main memory | Geometry (T & L) | N-ary arithmetic operators |
| List of vertices | Inner loop(s) of a code block | Blending functions | Reduction operators |
| Rendering passes | Outer loop of a code block | Clipping the scene | IF within the inner loop |
| Vertex indexing | First (inner) level of indirection | Active window | IF within intermediate loops |
| Textures lookup | Intermediate levels of indirection | Color index mask | IF within the outer loop |
| Color tables | Last (outer) level of indirection | Multipass rendering | Kernel programming |

Table 1
The GPU abstraction basics for a programmer.

are also investigated in our work, namely: (1) The memory allocation scheme for the input data, providing hints to OpenGL for an optimal data placement within the graphics card. (2) The new representation for color channels using 16-bit floating-point numbers (as introduced by NVIDIA in the GeForce 6 series), which allowed us to improve the accuracy in our results with little cost in execution time. (3) The driver impact on recent hardware developments, particularly PCI-Express.

The rest of this paper is organized as follows: Section 2 briefly introduces our methods for implementing general-purpose irregular algorithms within the GPU. Section 3 introduces the irregular kernels we use as benchmark. Section 4 describes alternatives for programming GPUs and discusses its influence in functionality and performance. Section 5 shows execution numbers for our benchmark compared with those of the CPU. Section 6 introduces our optimizations for reducing the graphics bus congestion. Section 7 summarizes related work, and finally Section 8 draws the conclusion from our work.

## 2. Our approach for mapping irregular computation onto the GPU

A typical graphics processor accepts an input stream (vertex attributes), transform it through a sequence of kernels or *shaders* (vertex program, fragment program, texture operators), and return an output stream (rasterized pixels), which is written into the frame buffer. Using GPUs for general-purpose computation entails disguising input data as vertex attributes, large data structures as textures, instructions as kernels, and final results as portions of video memory.

All these elements can be accessed by programmers using APIs such as DirectX or OpenGL. They just have to forget the traditional programming paradigm and focus on the data flow (the stream). Basically, each building block of a program constitutes a stream of vertices, whose geometry is defined according to existing loops and conditionals in the block for the kernels to compute only the desired elements. Multipass rendering executes the blocks sequentially, with the frame buffer and textures memory being used for the communication between consecutive blocks.

Table 1 shows a list of GPU-CPU equivalencies extracted from our experience when implementing codes on the graphics processor. More details on how to exploit data locality, map operators and implement indirect array accessing on the GPU can be found in [12,13]. Overall, GPUs are used for different purposes they are intended to, and our goal is to identify those valuable resources for irregular computation which can lead to a performance gain on the GPU. As it can be observed on the table, multiple level of indirections when accessing indexed arrays can be solved directly on the GPU hardware using vertices, textures and colors. In addition, we propose to implement reduction operators as blending functions to enhance performance versus the CPU.

Nonetheless, the GPU might well be seen more as a cooperator than a rival to the CPU, using executeAsync() calls to exploit task parallelism on a coarse grain algorithm decomposition: Since we deal with small kernels here, they might as well be considered as potential tasks the CPU delegates to the GPU as parts of a larger application.

| Data sets size | Small | Medium | Large |
|---|---|---|---|
| Euler nodes | 2800 | 9428 | 53961 |
| Euler edges | 17367 | 59863 | 353476 |
| File Size (Kb) | 576 | 1810 | 11524 |

| Matrix Rows | 3948 | 13992 | 28924 |
|---|---|---|---|
| Matrix nonzeros | 60882 | 316740 | 1036208 |
| Matrix Fill rate | 0.39 % | 0.16 % | 0.12 % |
| Matrix File Size | 1568 Kb | 2680 Kb | 8628 Kb |

Table 2
The input data set used for our benchmark. Left: Euler solver. Right: SpMxV.

## 3. The benchmark

We execute a couple of typical irregular kernels dealing with indirect array accessing:

1. The **Sparse Matrix-Vector Multiply (SpMxV)**, used as kernel in iterative methods within linear algebra. Indirections here are a consequence of the data structures used for storing matrix nonzeros in compressed formats (see Figure 1). The input vector, X, is stored as a texture, whereas Column is used as texture coordinates for accessing X, Data is stored as the color attribute for vertices, and Row lies in vertex positions defined to merge results onto the frame buffer holding Y.

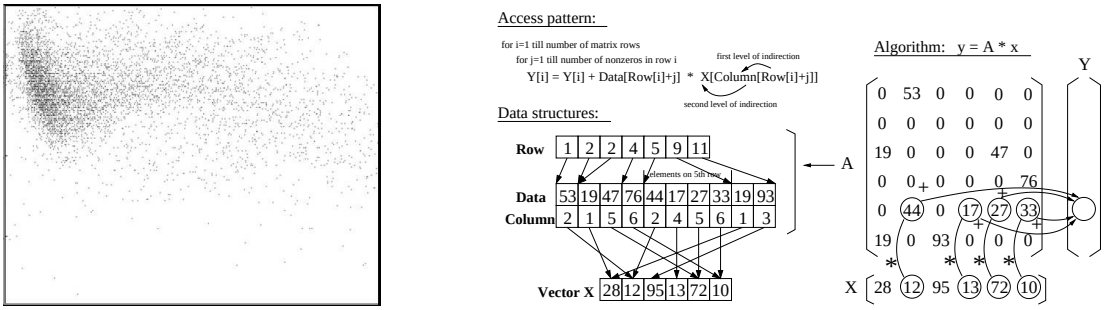



Figure 1. Data structures and access pattern for the sparse matrix-vector multiply (SpMxV).

2. **The Euler solver**, an adaptive partial different equation solver sweeping over unstructured meshes for discretizing complex domains and calculating forces between all pair of nodes connected through defined edges (see Figure 2). Major differences with respect to the SpMxV are the presence of indirections on the left hand side of assignments and the compact way for computing 3 statements in a vector manner using the RGB color space for the 3 components in the 3D force.

We decided in favour of iterative algorithms in order to perform a survey about the loading time into the GPU for the input data versus the number of iterations the GPU has to perform to amortize this cost. The AGP bus was already revealed as a potential bottleneck in former experiments, and so we selected applications where the computation/communication ratio might be high but at the same time variable, with the aim of testing its influence in GPU performance.

With a similar purpose, we selected input data sets of small, medium and large sizes for running the experiments (see Table 2). The left side of the table summarizes the features in three unstructured meshes used for the Euler solver, where nodes are connected through edges with an average connectivity of six. The input data set was taken from real applications at ICASE NASA Lab. The right side contains the parameters defining three different sparse matrices taken from the Harwell-Boeing collection, where they are represented in compressed row storage format.

## 4. Programming the GPU

### 4.1. Memory allocation

Using Vertex Buffer Objects (VBO) in OpenGL we were able to process vertices in custom ways without having to shuffle them between main memory and the card, something particularly useful

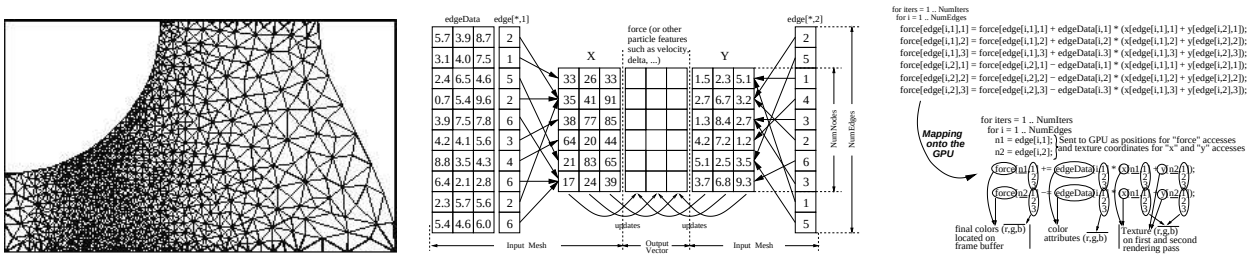



Figure 2. Data structures and access pattern for a 3D Euler kernel sweeping over an unstructured mesh.

| Year | CPU | Main Memory | GPU | Video Memory |
|---|---|---|---|---|
| 2003 | Pentium 4 @ 2.4 GHz | 1 Gb @ 4.2 Gb/s | GeF FX 5900 @ 450 MHz | 128 Mb @ 27.2 Gb/s |
| 2004 | Athlon 64 @ 2.0 GHz | 2 Gb @ 6.3 Gb/s | GeF FX5950U @ 475 MHz | 256 Mb @ 30.4 Gb/s |
| 2005 | Pentium 4 @ 3.2 GHz | 1 Gb @ 8.4 Gb/s | GeF 6800 GT @ 350 MHz | 256 Mb @ 35.2 Gb/s |

Table 3
Hardware features for our CPU-GPU comparison.

when a large amount of repeating geometry is involved in the GPU computation. This was always the case in our iterative algorithms, where access pattern (which defines the problem geometry for the GPU) remains unchanged through iterations. For further optimizations or when the buffer becomes too big for the memory available, we build block schemes switching between smaller portions of the buffer. This was carried out with the EXT_compiled_vertex_arrays OpenGL extension, which allowed us to lock and unlock vertex arrays for caching.

We started applying these schemes to the SpMxV code by strip-mining the loop sweeping over rows, and switching the Column vector between rows while keeping the X vector always within the VBO (its double indirection makes it to be the most unpredictable pattern). With all these optimizations, execution times for the SpMxV were 3-4 times faster, so we extended them to the Euler kernel as well, where the gains achieved were more modest. The ARB_Vertex_Buffer_Object extension was included in OpenGL 1.5, and later extended to pixels with the ARB_Pixel_Buffer_Object extension (December, 2004), an additional possibility we haven't tested in our experiments yet.

### 4.2. Floating-point precision

We tested the impact of computing reduction operators using the final blending stage in the graphics pipeline. In particular, the SpMxV algorithm performed the accumulation process of partial products this way (see right side on Figure 1), by reserving an area in the frame buffer for the result vector Y. Till the arrival of the GeForce 6 series, all these operations were poorly implemented on GPUs by using just 8-bit floating-point numbers associated to color representation in the RGB space.

This was a source of inaccuracy for the GPU within scientific computing, and remained to be seen whether computing on higher precision was going to hurt performance quite a bit. We measured the penalty for performing such operations using higher precision on a GeForce 6800 GT card versus an older GeForce 5950 Ultra model, and the execution times remained almost unaffected (roughly, those numbers correspond to the last two bars on each of the charts shown in Figure 3). We conclude that GPU is suffering from floating-point inaccuracy nowadays, but further developments towards 32-bit floating-point arithmetic will affect positively the precision without hurting performance severely, which is a good insight when thinking of GPUs as future general-purpose processors.

## 5. GPU performance versus CPU

Our OpenGL code with VBO and 16-bit floating-point precision in blending functions was compared against the CPU on regular PCs equipped with the latest CPUs and NVIDIA graphics cards from the GeForce 5 and 6 series.

On the CPU, we use Visual C++ 7.0 running under Windows XP. Multimedia extensions (SSE on Pentium 4 and 3DNow! Professional on Athlon 64) were enabled relying directly on HAL layer without any specific library in between. Cache performance was optimized sweeping consecutive memory addresses through loop reordering. Single-precision numbers were used as floating-point.

On the GPU, OpenGL 1.4 and 1.5 was used, plus a number of OpenGL extensions for subsequent optimizations: (1) **ARB_vertex_buffer_object** (Feb'03) allowed us an efficient memory allocation for vertices and colors onto the GPU (as discussed in Section 4.1). (2) **NV_texture_shader** (May'04) made it possible to tune the internal GPU texture shaders to our particular needs. (3) **NV_vertex_array_range** (Sep'01) and **NV_fence** (November 2003 - see [11]) allowed us to overlap CPU-GPU communications with GPU computation at different levels and switch vertices allocation between video and AGP memory. (4) **EXT_pbuffer** (Jan'99) enabled writing the results in areas different than the frame buffer, and reuse them in subsequent rendering passes. (5) **NV_float_buffer** (Jan'03) added floating-point support for textures, selecting their particular format as well as its association with color channels.

In addition, we used OpenGL interleaved arrays for sending vertex attributes to the GPU, vertex positions were precisely calculated according to screen resolution to skip interpolations, and GL_POINTS was selected as drawing primitive to keep computations strictly on the input list of vertices. Times in our benchmark were measured with the QueryPerformanceFrequency() and QueryPerformanceCounter() functions available in Visual C++. The nView tuning tool from NVIDIA was used to disable antialiasing, dithering and anisotropic filtering, mip filter and sample optimizations. Hardware acceleration was set to Single Display Mode and image setting was set to High Performance. Vertical synchronization was disabled since it limits the frame rate to the refresh rate of the particular monitor attached to the PC.

### 5.1. Execution time

Execution times for the CPUs and GPUs are shown in Figure 3, with the upper row corresponding to the Euler kernel and the lower row to the SpMxV code. On each chart, the three bars on the left are CPU times; the ones on the right belong to GPUs. In order to perform a fair comparison, first and fourth bars correspond to the same PC, and so do second and fifth as well as third and sixth.

It can be seen that performance is between a 2-4 factor in favour of GPUs, and that the difference is increasing with the size of the data set. However, the burden for loading the input vertices and textures increases with the data set as well (see Figure 4.a)

### 5.2. Communication time

PCI-Express replaces the parallel multidrop architecture of AGP with a serial, point-to-point connection bus [2]. Current bandwidth reaches 4 Gbytes/s., doubling AGP 3.0 plus an extra factor of two since PCI-Express is a dual-simplex specification willing to communicate both ways simultaneously. Besides, PCI-Express frequency can still be increased up to 10 gigatransfers per second from the current value of 2.5, and bus width can be doubled to 32x from the actual 16x implementation.

Figure 4.a illustrates PCI-express performance when used for loading data in our benchmarks. As compared to AGP, the overhead was reduced by around 40% on the larger data set. Our results show that AGP loading times represent roughly 20 times the execution time, whereas on PCI-Express they

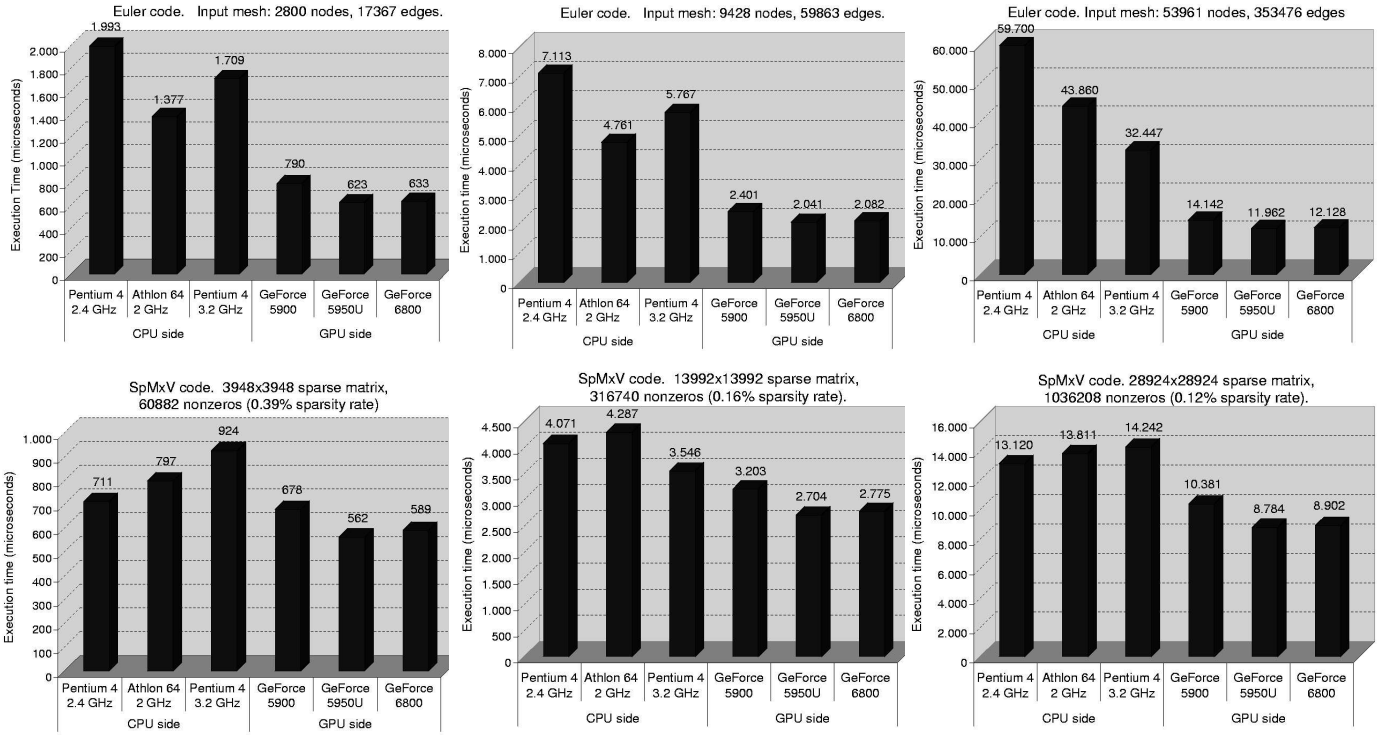



Figure 3. CPU-GPU comparison in execution time for the Euler and SpMxV codes (excluding loading times).

come down to nearly 12 times. Moreover, without PCI-Express the Euler code has to execute seven iterations for the GPU to defeat the CPU on the larger execution when accounting the communication time. Using PCI-Express, this requirement is cut to only four iterations. For the SpMxV kernel, the number of iterations required to amortize this cost are 85 for AGP, and 55 for PCI-Express.

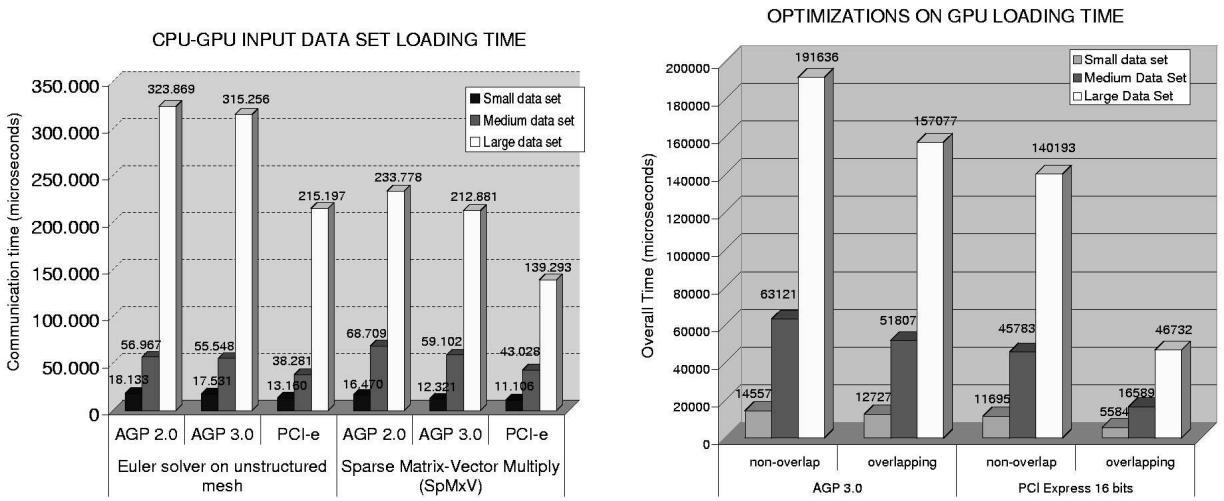
The huge difference between the two codes lies in the compact manner that Euler describes the geometry: A single (x,y,z) position and (s,r) texture coordinate is shared among the 3 components of the Force, X and Y arrays. This is opposed to the SpMxV code, where the size of the communication buffer is 8 times the number of nonzeros (for each vertex, we have (s,r), (r,g,b) and (x,y,z)).

## 6. Overlapping communication/computation

Computation time comprises the actual time that data is processed on the GPU; loading time includes the tasks for converting the input data sets into graphical data structures and passing them onto the GPU. To speed-up the entire process while decoupling the GPU computation from its communication needs, we use the NVIDIA OpenGL extension NV_vertex_array_range to place vertices directly onto GPU accessible memory. Then, we overlap communication and computation using the NV_fence extension, a fine grained synchronization mechanism available in OpenGL [11].

We allocate a circular buffer in video memory and partition it into several smaller buffers. The CPU now places a part of the vertex data into the first partition, gives the call to the GPU to process this data, and while the GPU pulls and computes the data, the CPU places another part of the vertex data for the next partition and so on.

Figure 4.b shows the results we obtained when performing this optimization over the SpMxV code. Due to the overlapping, we cannot split loading and execution time, nor decouple one iteration from another. Instead, we performed 100 SpMxV iterations and then accounted for all the commu-




(a) (b)

Figure 4. (a) AGP/PCI-Express comparison for loading the small, medium and large meshes (Euler) and sparse matrices (SpMxV). (b) Overall execution time (loading times plus a single rendering pass) for the SpMxV when overlapping CPU communications with GPU communications on different matrix sizes.

nication time that was not overlapped with computation, taking that as the actual communication overhead. Loading times were this way greatly reduced, roughly 20% for the AGP bus and almost 70% on PCI-Express, where snooping was revealed as an outstanding mechanish for maximizing the desired overlap. We also measured the effect of having more recent drivers in the PCI-Express machine than in the AGP PC, finding out the drivers to be responsible for just marginal gains.

## 7. Related Work

The use of graphics hardware for executing general-purpose applications is becoming increasingly popular ([1]). Some examples outperforming CPUs are volume segmentation (10-20 times faster) [10], surfaces deformation (10-15x) [9], multigrid solvers (3x) [6] and linear algebra (2x) [3,8]. In all those cases, the operations are expressed in terms of appropriate graphics operators, though the output was first constrained to integers and later to low precision floating-point data.

Currently, 32-bit precision is supported on several graphics cards, but some constraints still remain at the API level: In OpenGL, stages like rasterization clamp vertex attributes to the (0..1) range unless you use programmable shaders to bypass those operations, and blending functions or texture handling can process only 16-bit floating-point numbers, which is the most sofisticated color representation up to date. Besides, DirectX doesn't allow you to specify data 32-bit long in certain operations. The results we show through this paper are also affected by such limitations.

Data reuse and building computational blocks on the GPU was also an aspect recently investigated by Fatahalian et al. [5], who implemented a dense matrix-matrix multiplication on the GPU using programmable shaders to conclude that the lack of cache memory will limit GPU performance. We don't find cache that valuable when the access pattern is irregular, and besides we prefer to avoid the use of shaders because that would force us to decompose the problem using multirendering.

Even though shaders restrictions may be relaxed in the future, it means computing the access indices at run-time, a major burden when indirections predominate. In cases like the SpMxV, where the access pattern remains constant through iterations, we set up the geometry for vertex position to act as a tag for guiding the streaming computation, which qualify us for extracting the entire index calculation out of the execution loops and amortizing the loading time through iterations.

## 8. Conclusions

In this paper, we implement a couple of irregular computational kernels onto the graphics pipeline, showing how general-purpose applications can benefit from a streaming execution model to outperform current CPUs by a wide margin . Our methods avoid programming the shaders to overcome their current limitations, and benefit from a bunch of OpenGL extensions to compute the whole algorithm on a single rendering pass.

We focus this work on the experimental side at different levels of the graphics card: (1) API: OpenGL turned out to be faster than DirectX, and also offered us much richer extensions for further optimizations. (2) Memory allocation: Vertex Buffer Objects became an efficient mechanism for reusing data geometry through iterations and building computational blocks. (3) Floating-point precision was not hurting performance when enhanced from 8-bit to 16-bit in the final stages of the graphics pipeline, as already available in the GeForce 6 Series by Nvidia. (4) Communication time accounted for most of the running time and was vastly reduced, first 40% using PCI-Express and then up to an additional 70% when overlapping with computational times through a large number of iterations. Further improvements will be reached when PCI-Express shows its extraordinary scalability in the future, removing the actual bottleneck from the graphics card.

Driven by the game industry, future graphics cards are expected to continue increasing their capabilities so that current restrictions on shaders, floating-point accuracy and communication overhead will be relaxed and virtually any application can be mapped onto the graphics pipeline.

## References


[1] A Web page dedicated to the latest developments in general-purpose on the GPU. http://www.gpgpu.org.

[2] A.V. Bhatt. *Creating a Third Generation I/O Interconnect.* Intel Developer Network for PCI Express.

[3] Bolz, J., Farmer, I., Grinspun, E., Schroder, P. *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid*. Procs. (SIGGRAPH'03). San Diego (California). July, 2003.

[4] Duff I.S., Grimes R.G., and Lewis, J.G. *User's guide for the Harwell–Boeing sparse matrix collection (Release I).* Technical Report TR/PA/92/86, CERFACS, Toulouse, 1992.

[5] K. Fatahalian, J. Sugerman, and P. Hanrahan. *Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication.* Procs. (HWWS'04). Grenoble (France), August, 2004.

[6] Goodnight, N., Woolley, C., Lewin, G., Luebke, D., and Humphreys, G. *A multigrid solver for boundary value problems using programmable graphics hardware*. Procs. (HWWS'03). pp.102-111, July, 2003.

[7] Khailany, B., Dally, W., Rixner, S., Kapasi, U., Owens, J. and Towles, B. *Exploring the VLSI Scalability of Stream Processors*. Procs. 9th Symposium on High Performance Computer Architecture. Anaheim (California), Feb. 2003, pp. 153-164.

[8] Kruger, J., Westermann, R. *Linear Algebra Operators for GPU Implementation of Numerical Algorithms*. Procs. (SIGGRAPH'03). San Diego (California). July, 2003.

[9] Lefohn, A., Kniss, J., Hansen, C., and Whitaker, R. *Interactive Deformation and Visualization of Level Set Surfaces Using Graphics Hardware*. Procs. 14th IEEE Visualization Conference, Seattle (Washington), October, 2003, pp. 75-82.

[10] Sherbondy, A., Houston, M., Napel, S. *Fast Volume Segmentation With Simultaneous Visualization Using Programmable Graphics Hardware*. Procs. 14th IEEE Visualization Conf., Seattle, October, 2003.

[11] Spitzer, J. and Everitt, C. GL_NV_vertex_array_range and GL_NV_fence on GeForce Products and beyond. NVIDIA Corporation. August, 2001.

[12] M. Ujaldón, J. Saltz. *Mapping Irregular Computation onto the Graphics Pipeline.* Internal Report 2004, Biomedical Informatics Dept, Ohio State University.

[13] M. Ujaldón, J. Saltz. *The GPU as an indirection engine for a fast information retrieval.* Intl J. Electronic Business, Ed. Inderscience, vol. 3, number 3/4, pages 316-327. July-August, 2005.